Towards a formal account for software transactional memory

Doriana Medić, Claudio Antares Mezzina, Iain Phillips, Nobuko Yoshida

Towards a formal account for software transactional memory*

Doriana Medić[1], Claudio Antares Mezzina[2], Iain Phillips[3], and Nobuko Yoshida[3]

[1] Focus Team, University of Bologna/Inria, France
[2] Dipartimento di Scienze Pure e Applicate, Universitá di Urbino, Italy
[3] Imperial College London, United Kingdom

Abstract. Software transactional memory (STM) is a concurrency control mechanism for shared memory systems. It is opposite to the lock based mechanism, as it allows multiple processes to access the same set of variables in a concurrent way. Then according to the used policy, the effect of accessing to shared variables can be committed (hence, made permanent) or undone. In this paper, we define a formal framework for describing STMs and show how with a minor variation of the rules it is possible to model two common policies for STM: reader preference and writer preference.

Keywords: STM · Transactions · Concurrency.

1 Introduction

Starting from the 1960s, reversible computing has been studied in several contexts ranging from quantum computing [6], biochemical modelling [7], programming [8,9], and program debugging [15,10]. Distributed reversible actions can be seen as defeasible partial agreements: the building blocks for different transactional models and recovery techniques. The work of Danos and Krivine on reversible CCS (RCCS) [1] provides a good example: they show how notions of reversible and irreversible actions in a process calculus can model a primitive form of transaction, an abstraction that has been found useful, in different guises, in reliable concurrent and distributed programming. Since the seminal work of [1], other works have investigated the interplay between transactions and reversibility [2,11] in the area of message passing systems. On the shared memory side, we just recall the work of [12] where a CCS endowed with a mechanism for software transactional memories (STMs) is presented. Another work about reversibility and a high-level abstraction of shared memory (tuple spaces) is presented in [16].

* This work has been partially supported by French ANR project DCore ANR-18-CE25-0007 and by the Italian INdAM – GNCS project 2020 *Reversible Concurrent Systems: from Models to Languages*. We also acknowledge partial support from the following projects: EPSRC EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833, EP/N028201/1, and EP/T006544/1.

Software Transactional Memory [3,4] is an elegant way to address the problem of concurrent programming, by relieving the programmer from the burden of dealing with locks. The lock-based approach is error prone and usually leads to deadlocks when the complexity of the system grows. Opposite to the lock-based approach, STM uses transactions. A transaction is a block of code accessing shared data which is meant to be executed *atomically* with an "all or nothing" policy: that is either all the effects of a transaction have to be visible when it commits, or none of them has to be visible in case of abortion. This abstraction allows for multiple transactions to be executed "at the same time". The programmer just needs to specify the sequences of operations to be enclosed in transactions, while the system is in charge of the interleaving between the concurrent transactions. A transaction can either *commit* and update the system permanently or *abort* and discard all the changes done by its execution.

In this work, we are interested in the interplay between reversible computing and the STM approach to control the concurrent executions. Therefore, we present a formal framework for describing STMs in a simple shared memory context. In particular, when a transaction aborts, it is necessary to discard all the updates that it made and we need to bring the system back to the state before the execution of the transaction. To accomplish the behaviour above, we implement a rollback operator following the approach given in [13]. A transaction can access a shared variable either in read or in write mode. Given this, different policies can be used to regulate the transactions which are accessing the same value in the shared memory. According to the implemented policy, some transactions will succeed and some will be aborted. We will show how it is possible to model *writer* and *reader preference* [5] in our framework. Consider the following C-like code where two functions/threads access the same shared variables:

```
int x = 0;
int y = 5;
int z = 0;
```

```
void t1()
 {z = y+x;  }
void t2()
 { x = z+1; }
```

t1 ; t2	t2 ; t1	t1 | t2
z = 5	z = 6	z = 5
x = 6	x = 1	x = 1

All the possible executions of the two functions are reported above: either the two functions are executed sequentially or are interleaved (leading to an unwanted state). If we wrap the two functions into two atomic blocks then the third behaviour would be automatically ruled out by the system as one of the two transactions will be aborted depending on the implemented policy.

2 Syntax

In this section we give the syntax of our calculus. Let us assume the existence of the mutually disjoint sets $\mathcal{V}$ (a set of variables) and $\mathcal{I}$ (a set of transaction identifiers), ranged over by x, y, z and t, h, respectively.

The syntax of the calculus is given in Figure 1. *Write* and *read* access to the variable x are represented with actions $\mathtt{wr}(x)$ and $\mathtt{rd}(x)$. The sequential

$$
\begin{array}{rlll}
\text{(Actions)} & \alpha, \beta & ::= & \mathtt{wr}(x) \mid \mathtt{rd}(x) \\
\text{(Processes)} & A, B & ::= & \mathbf{0} \mid \sum_i \alpha_i.A_i \\
\text{(Expressions)} & X, Y & ::= & B \mid \alpha.X \mid X;Y \mid \ (X \mid Y) \ \mid t : [\![A]\!]_\Gamma \\
\text{(Configuration)} & C & ::= & X \parallel M \\
\text{(Shared Memory)} & M & ::= & \langle x, W, R\rangle \parallel M
\end{array}
$$

Fig. 1. Syntax

execution of the actions $\mathtt{wr}(x)$ and $\mathtt{rd}(x)$ together with the choice operator $+$ build the *processes*, given with A, B productions. The term $t : [\![A]\!]_\Gamma$ represents a *transaction*, where t is a unique identifier, A is the body of the transaction and Γ is the set recording the identifiers of the transactions which have the write access to the variable that transaction t has to read. The idea behind the set Γ is to allow transaction t to have read access to any variable, but to record the write access to them. In this way if the transaction that writes on the variable fails, the transaction that reads the same variable has to fail too. More explanations will be given in Section 3.

Transactions, together with processes, build *expressions*. An expression can be prefixed with the actions $\mathtt{wr}(x)$ and $\mathtt{rd}(x)$ and we denote it as $\alpha.X$. Two expressions X and Y can be executed in parallel, $X \mid Y$, or in sequential order $X;Y$. We can note that the expression X can be the process that is not inside of the transaction, and that operation ; allows us to have a transaction followed by an action (for example $t : [\![A]\!]_\Gamma; \mathtt{wr}(x)$). The whole system, called *configuration*, is denoted with C and it represents the expressions together with the *shared memory*. The shared memory M is made of triples of the form $\langle x, W, R\rangle$ for every variable in the system. In $\langle x, W, R\rangle$, x is the variable name, W and R are the sets recording transactions which had write and read access to x, respectively. Let us note that we abstract away from the value contained by variables, since this is not relevant for our framework. We just need to record whether a variable is read (a transaction reads its value) or modified (a transaction changes its value).

In order to write expressions in a more compact way, we define the notion of *history* context. For instance, having a transaction $t : [\![\mathtt{wr}(x).\mathtt{rd}(x_1)\mathtt{rd}(y).A + B]\!]_\Gamma$ we can write it as $t : [\![\mathtt{H}[\mathtt{rd}(x_1)\mathtt{rd}(y).A]]\!]_\Gamma$ where $\mathtt{H} = \mathtt{wr}(x).\bullet + B$. Formally:

Definition 1 (History context). *A history context* $\mathtt{H}$ *is a process with a hole* $\bullet$*, defined by the following grammar:* $\mathtt{H} ::= \bullet \mid \alpha.\bullet + A$.

3 Semantics

The semantics of our calculus is presented in two steps. First, we give the basic rules of the framework (common to all the policies) and then, we present the extra rules, necessary to model *reader* or *writer preference*. With *reader preference*,

we intend that reading the value of a variable is always possible, i.e. no read access should be suspended, unless the write access already took place. *Writer preference*, on the other side, allows the write access to the value of a variable x even if some read access already took place. In this case, all the executing transactions with the read access to a value x need to be aborted and brought back to their initial state.

In what follows we provide the auxiliary functions necessary for the semantics of the calculus: the function which computes the set of the transaction identifiers of a given expression and the operation which removes transaction identifiers from the system.

Definition 2 (Set of the transaction identifiers). *The set of the transaction identifiers of a given expression X, written $\mathtt{id}(X)$, is inductively defined as:*

$$\mathtt{id}(Y \mid Y') = \mathtt{id}(Y) \cup \mathtt{id}(Y') \qquad \mathtt{id}(\alpha.Y) = \mathtt{id}(Y) \qquad \mathtt{id}(A) = \emptyset$$
$$\mathtt{id}(Y; Y') = \mathtt{id}(Y) \cup \mathtt{id}(Y') \qquad \mathtt{id}(t : [\![A]\!]_\Gamma) = \{t\}$$

Definition 3 (Removing of identifiers). *The operation of deleting transaction identifier t from the configuration C, denoted as $C_{@t}$, is defined as follows:*

$$(X \parallel M)_{@t} = X_{@t} \parallel M_{@t} \qquad (\alpha.X)_{@t} = \alpha.(X_{@t})$$
$$(X \mid Y)_{@t} = X_{@t} \mid Y_{@t} \qquad (t' : [\![A]\!]_\Gamma)_{@t} = t' : [\![A]\!]_{\Gamma \setminus t}$$
$$(X; Y)_{@t} = X_{@t}; Y_{@t} \qquad (\langle x, W, R\rangle \parallel M)_{@t} = \langle x, W \setminus t, R \setminus t\rangle \parallel M_{@t}$$

When a transaction fails, the effects of the internal computation are undone and the entire transaction is restarted, that is, brought back to its initial state. As a consequence, the transactions depending on it are also rolled back. Dependency between transactions changes with the chosen preference. We shall see more information about the preferences by the end of this section.

To be able to identify the state of the internal computation of a transaction, we mark it with symbol $^\wedge$. For instance, if we consider transaction $t : [\![\mathtt{rd}(x).\mathtt{rd}(y).{}^\wedge\mathtt{wr}(z).\mathtt{wr}(x')]\!]_\Gamma$, the actions $\mathtt{rd}(x)$ and $\mathtt{rd}(y)$ represent the past of the transaction and the action $\mathtt{wr}(z)$ is the next action to be executed.

Now we define our rollback operator which brings a transaction back to its initial state i.e. the symbol $^\wedge$ is placed in the beginning of the transaction and its set Γ is empty. For instance, if we roll back transaction $t : [\![\mathtt{rd}(x).\mathtt{rd}(y).{}^\wedge\mathtt{wr}(z).\mathtt{wr}(x')]\!]_\Gamma$, we obtain $t : [\![{}^\wedge\mathtt{rd}(x).\mathtt{rd}(y).\mathtt{wr}(z).\mathtt{wr}(x')]\!]_\emptyset$. Formally, we have:

Definition 4 (Rollback operator). *The rollback operator on the transaction $t : [\![A]\!]_\Gamma$, written* $\mathbf{roll}(t)$*, is defined as:* $\mathbf{roll}(t) = t : [\![{}^\wedge A]\!]_\emptyset$.

In what follows, we give the semantics of our calculus. We shall start by introducing semantics rules representing the base of our framework (rules that are common for both models) and then we show the additional rules for each preference.

The common rules are given in Figure 2. An action executed outside a transaction can be seen as an atomic step in which the action is discarded after the

$$(\textsc{WriteP})\ \mathtt{wr}(x).A + B \parallel \langle x, \emptyset, \emptyset\rangle \rightarrow\ A \parallel \langle x, \emptyset, \emptyset\rangle$$

$$(\textsc{ReadP})\ \mathtt{rd}(x).A + B \parallel \langle x, \emptyset, \emptyset\rangle \rightarrow\ A \parallel \langle x, \emptyset, \emptyset\rangle$$

$$(\textsc{Write})\ \frac{(W \subseteq \{t\} \wedge R \subseteq \{t\})}{t : [\![\mathtt{H}[^\wedge \mathtt{wr}(x).A + B]]\!]_\Gamma \parallel \langle x, W, R\rangle \parallel M \rightarrow t : [\![\mathtt{H}[\mathtt{wr}(x).^\wedge A + B]]\!]_\Gamma \parallel \langle x, W \cup t, R\rangle \parallel M}$$

$$(\textsc{Read})\ t : [\![\mathtt{H}[^\wedge \mathtt{rd}(x).A + B]]\!]_\Gamma \parallel \langle x, W, R\rangle \parallel M \rightarrow t : [\![\mathtt{H}[\mathtt{rd}(x).^\wedge A + B]]\!]_{\Gamma \cup (W \setminus t)} \parallel \langle x, W, R \cup t\rangle \parallel M$$

$$(\textsc{Par})\ \frac{X \parallel M \rightarrow X' \parallel M' \quad \mathtt{id}(X) \cap \mathtt{id}(Y) = \emptyset}{X \mid Y \parallel M \rightarrow X' \mid Y \parallel M'}$$

$$(\textsc{Commit})\ \frac{t : [\![A^\wedge]\!]_\Gamma; Y \mid X \parallel M \quad \wedge \quad \Gamma = \emptyset}{t : [\![A^\wedge]\!]_\Gamma; Y \mid X \parallel M \rightarrow Y \mid X_{@t} \parallel M_{@t}}$$

$$(\textsc{RollR})\ \frac{t : [\![A]\!]_\Gamma \parallel M \rightarrow \mathbf{roll}(t) \parallel M_{@t} \quad \forall t_i\ \ t_i : [\![A_i]\!]_{\Gamma_i \cup \{t\}}}{t : [\![A]\!]_\Gamma \mid \prod_i t_i : [\![A_i]\!]_{\Gamma_i} \parallel M \rightarrow \mathbf{roll}(t) \mid \prod_i \mathbf{roll}(t_i) \parallel (M_{@t})_{@t_i}}$$

Fig. 2. Common rules for both models

execution (rules WriteP and ReadP). Therefore, there is no need to keep track of its access to the variable. The only constraint is that they cannot access the variable while some transaction has read or write access to it.

Rule Write describes when a transaction can modify the content of a variable. To do so, there should not be another transaction which has already accessed the variable in either writing or reading mode. After the execution, the identifier t is added to the write access set W of the variable x and the symbol $^\wedge$ is moved to the next computational step. Rule Read allows the transaction t to execute the action $\mathtt{rd}(x)$ at any moment. Then the identifier t is added to the set R of the variable x and the set $W \setminus t$ is added to the set Γ (if write and read access to the variable x are in the same transaction t, then it is not necessary to save the identifier t into a set Γ).

To have a better intuition about these two rules, we give a simple example. Consider the transaction t with a corresponding shared memory

$$t : [\![^\wedge \mathtt{wr}(x).\mathtt{rd}(y)]\!]_\emptyset \parallel \langle x, \emptyset, \emptyset\rangle \parallel \langle y, \emptyset, \emptyset\rangle$$

After executing the write access to the variable x, we obtain the system

$$t : [\![\mathtt{wr}(x).^\wedge \mathtt{rd}(y)]\!]_\emptyset \parallel \langle x, \{t\}, \emptyset\rangle \parallel \langle y, \emptyset, \emptyset\rangle$$

where the pointer $^\wedge$ is moved to the next action and the identifier t is added to the write set of the variable x. Now we can perform the read access to variable y and we have:

$$t : [\![\mathtt{wr}(x).\mathtt{rd}(y)^\wedge]\!]_\emptyset \parallel \langle x, \{t\}, \emptyset\rangle \parallel \langle y, \emptyset, \{t\}\rangle$$

where the identifier t is added to the read set of the variable y. The set Γ of the transaction t remains empty since there is no transaction which had write access to variable y.

Rule Par allows expressions to execute in parallel (in an interleaving fashion) ensuring the uniqueness of the identifier t. By executing its last action, the

transaction t can commit if the set Γ is empty, by applying the rule Commit. After it commits, the execution proceeds with the continuation Y and the identifier t is deleted from the remaining system. The intuition is that transaction t can commit if the other transactions, having a write access to the variables that transaction t read, have been committed. The rollback of the transaction t can be done with the rule RollR. It will force every transaction in parallel having the identifier t in their set Γ to roll back too. The intuition is that when the transaction with $\mathtt{wr}(x)$ rolls back, every transaction which has read access to x should roll back as well. For instance, let us consider the system containing following transactions:

$$t : [\![A]\!]_{\Gamma} \mid t_1 : [\![A_1]\!]_{\{t\}} \mid t_2 : [\![A_2]\!]_{\Gamma_2} \qquad \text{such that } \mathtt{rd}(x) \in A_1 \text{ and } t \notin \Gamma_2$$

and that transaction t needs to be rolled back. Then, by applying the rule RollR, we obtain the system:

$$\mathbf{roll}(t) \mid \mathbf{roll}(t_1) \mid t_2 : [\![A_2]\!]_{\Gamma_2}$$

in which transaction t_1 is rolled back too since $t \in \Gamma_1$ while t_2 remains the same.

Now we can give the rules necessary to model *reader* and *writer* preference. To give a better intuition about the differences between the two models, we use the example from the introduction as a running example.

Reader preference. To model the reader preference we use the rules from Figure 2 and the rule given below.

$$\text{(R-RollW)}\ \frac{(W \not\subseteq \{t\} \vee R \not\subseteq \{t\})}{t : [\![\mathtt{H}[\hat{}\,\mathtt{wr}(x).A]]\!]_{\Gamma} \parallel \langle x, W, R\rangle \parallel M \to \mathbf{roll}(t) \parallel \langle x, W, R\rangle_{@t} \parallel M_{@t}}$$

The rollback operator is triggered when the transaction t cannot write on the variable x (this happens when $W \not\subseteq \{t\}$ or $R \not\subseteq \{t\}$). With the rule R-RollW the transaction t goes to the state $\mathbf{roll}(t)$, i.e. the initial state of the transaction. Additionally, the identifier t is removed from every triple of the shared memory.

To illustrate it, we use the example from the introduction, abstracting away from the read and write values contained in variables and representing accesses of two threads to the shared memory in our framework with transactions t_1 and t_2. Transaction t_1 has read accesses to variables y and x and then writes on variable z, while t_2 has read access to variables z and then writes on x. We have the following system

$$t_1 : [\![\hat{}\,\mathtt{rd}(y).\mathtt{rd}(x).\mathtt{wr}(z)]\!]_{\emptyset} \mid t_2 : [\![\hat{}\,\mathtt{rd}(z).\mathtt{wr}(x)]\!]_{\emptyset} \parallel \langle x, \emptyset, \emptyset\rangle \parallel \langle y, \emptyset, \emptyset\rangle \parallel \langle z, \emptyset, \emptyset\rangle$$

We assume that read accesses are executed in parallel and the obtained system is

$$t_1 : [\![\mathtt{rd}(y).\mathtt{rd}(x).\hat{}\,\mathtt{wr}(z)]\!]_{\emptyset} \mid t_2 : [\![\mathtt{rd}(z).\hat{}\,\mathtt{wr}(x)]\!]_{\emptyset} \parallel \langle x, \emptyset, \{t_1\}\rangle \parallel \langle y, \emptyset, \{t_1\}\rangle \parallel \langle z, \emptyset, \{t_2\}\rangle$$

Now transaction t_1 is executing write access to variable z but since in the memory for variable z we have $R \not\subseteq \{t_1\}$ ($R = \{t_2\}$), the transaction t_1 needs to roll back according to the rule R-RollW, and we have

$$\mathbf{roll}(t_1) \mid t_2 : [\![\mathtt{rd}(z).\hat{}\,\mathtt{wr}(x)]\!]_{\emptyset} \parallel \langle x, \emptyset, \emptyset\rangle \parallel \langle y, \emptyset, \emptyset\rangle \parallel \langle z, \emptyset, \{t_2\}\rangle$$

where $\mathbf{roll}(t_1) = t_1 : [\![\hat{}\,\mathtt{rd}(y).\mathtt{rd}(x).\mathtt{wr}(z)]\!]_{\emptyset}$.

Writer preference. To model the writer preference we use the rules from Figure 2 and the rules given below.

$$\text{(W-PREF)}\ \frac{W \subseteq \{t\} \quad \wedge \quad R \not\subseteq \{t\} \quad \wedge \quad R' = R \setminus t}{t : [\![\mathtt{H}[^\wedge \mathtt{wr}(x).A + B]]\!]_\Gamma \mid \prod_{t_i \in R'} t_i : [\![A_i]\!]_{\Gamma_i} \parallel \langle x, W, R\rangle \parallel M \to t : [\![\mathtt{H}[\mathtt{wr}(x).^\wedge A + B]]\!]_\Gamma \mid \prod_{t_i \in R'} \mathbf{roll}(t_i) \parallel \langle x, W \cup t, R\rangle_{@t_i} \parallel M_{@t_i}}$$

$$\text{(W-ROLLW)}\ \frac{(W \not\subseteq \{t\})}{t : [\![\mathtt{H}[^\wedge \mathtt{wr}(x).A]]\!]_\Gamma \parallel \langle x, W, R\rangle \parallel M \to \mathbf{roll}(t) \parallel \langle x, W, R\rangle_{@t} \parallel M_{@t}}$$

The rollback is triggered by the writer only in the case when another transaction has write access to the same variable. Therefore the condition on the rule W-ROLLW is simply $W \not\subseteq \{t\}$. The additional rule, with respect to the reader preference is the rule W-PREF. It allows a transaction to modify the value of a variable x if other transactions have read access to it. At the same time, all transactions executing in parallel whose identifiers belong to the set R, are requested to roll back.

To illustrate it, we use the same example as for the reader preference where read accesses are executed already. Therefore, we have the system

$$t_1 : [\![\mathtt{rd}(y).\mathtt{rd}(x).^\wedge \mathtt{wr}(z)]\!]_\emptyset \mid t_2 : [\![\mathtt{rd}(z).^\wedge \mathtt{wr}(x)]\!]_\emptyset \parallel \langle x, \emptyset, \{t_1\}\rangle \parallel \langle y, \emptyset, \{t_1\}\rangle \parallel \langle z, \emptyset, \{t_2\}\rangle$$

Now we can execute the write access to variable z, since in the rule W-PREF the condition for the read set R allows a transaction to perform the write access, and in that case all transactions in parallel having read access to variable z need to be rolled back. Therefore, transaction t_1 executes write access, while t_2 will be rolled back, and we have:

$$t_1 : [\![\mathtt{rd}(y).\mathtt{rd}(x).\mathtt{wr}(z)^\wedge]\!]_\emptyset \mid \mathbf{roll}(t_2) \parallel \langle x, \emptyset, \{t_1\}\rangle \parallel \langle y, \emptyset, \{t_1\}\rangle \parallel \langle z, \{t_1\}, \emptyset\rangle$$

where $\mathbf{roll}(t_2) = t_2 : [\![^\wedge \mathtt{rd}(z).\mathtt{wr}(x)]\!]_\emptyset$.

4 Conclusion and Future Work

We have presented a framework to express the STM mechanism in a simple shared memory context. The framework is able to model two different policies for the execution of the concurrent transactions: writer and reader preference. Our intention is to start from a simple calculus and then to add in a modular way: nested transactions, data structures (e.g., C structures) and more complex scheduling policies. Nested transactions will require to record for each transaction a list of its *children* transactions. These children inherit the access of the parent transaction. There exist different policies to deal with nested transactions: *closed* nested transactions [17] and open nested transactions [18]. The difference is that in the first case the parent does not execute till all the children have committed, while in the second case the parent can commit even before its children. This may lead to inconsistencies which have to be dealt with compensations.

Our ultimate goal is then to prove that the modular framework satisfies the *opacity* [14] property, that is, all the execution traces of our semantics, where the transactional bodies are interleaved, are equivalent to executions in which transactional blocks are executed as a whole (in a lock-based fashion) without being interleaved with other transactions.

References

1. V. Danos and J. Krivine. Transactions in RCCS. In: CONCUR - Concurrency Theory, LNCS, vol 3170, pp 39–412, San Francisco (2005).
2. I. Lanese, M. Lienhardt, C.A. Mezzina, A. Schmitt and J-B. Stefani. Concurrent Flexible Reversibility. In: ESOP, LNCS, vol 7792, pp 370–390, Italy (2013).
3. M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In: Annual International Symposium on Computer Architecture, pp 289–300, ACM, San Diego (1993).
4. N. Shavit, D. Touitou. Software transactional memory. In: PODC, pp 204–213. ACM, New York (1995).
5. P-J. Courtois, F. Heymans and D. L. Parnas. Concurrent Control with "Readers" and "Writers". In: Commun. ACM 1971, vol.14, pp 667–668.
6. J. Grattage. A functional quantum programming language. In LICS 2005, pp 249–258. IEEE Computer Society, Washington (2005).
7. I. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In: RC, LNCS, vol 7581, pp 21–232, Denmark (2012).
8. C. Lutz. Janus: a time-reversible language. Letter to R. Landauer., 1986.
9. T. Yokoyama, H. B. Axelsen and R. Glück. Principles of a reversible programming language. In: Conference on Computing Frontiers, ACM, pp 43–54, Italy (2008).
10. I. Lanese, N. Nishida, A. Palacios, and G. Vidal. Cauder: A causal-consistent reversible debugger for Erlang. In: FLOPS, LNCS, vol 10818, pp 24–263, Japan (2018).
11. E. de Vries, V. Koutavas, M. Hennessy. Communicating Transactions. In: CONCUR 2010. LNCS, vol 6269, pp 56–583, Heidelberg (2010).
12. L. Acciai, M. Boreale and S. Dal-Zilio. A Concurrent Calculus with Atomic Transactions. In: ESOP, LNCS, vol 4421, pp 48–63, Portugal (2007).
13. I. Lanese, C. A. Mezzina, A. Schmitt and J-B. Stefani. Controlling Reversibility in Higher-Order Pi. In: CONCUR 2011, LNCS, vol 6901, pp 297–311, Germany (2011).
14. R. Guerraoui, M. Kapalka: On the correctness of transactional memory. In: PPOPP, pp 175–184. ACM, New York (2008).
15. E. Giachino, I Lanese and C. A. Mezzina: Causal-Consistent Reversible Debugging. In: FASE 2014, pp 370–384. LNCS, vol 8411, France (2014).
16. E. Giachino, I. Lanese, C. A. Mezzina and F. Tiezzi: Causal-consistent rollback in a tuple-based language. In: JLAMP, vol 88, pp 99–120, (2017).
17. J. Gray and A. Reuter: Transaction Processing: Concepts and Techniques. (1993).
18. J. E. B. Moss: Open Nested Transactions: Semantics and Support. (2006).